University of Southern Denmark

DM852

Heuristics & Approximations Algorithms

Submitted To:
Marco Chiarandini
Lene Monrad Favrholdt
IMADA
Mathematics & Computer Science Department

Submitted By :
Alexander Lerche Falk
Narongrit Unwerawattana
Spring - Master of Computer Science

Contents

1 Introduction

This project shows custom heuristic algorithms, resolving the Capacitated Vehicle Routing Problem (CVRP). When the heuristics have resolved the problem, given a canonical solution, we are applying custom and known Local Search algorithms to improve the results from the heuristics. The CVRP gives insight into how to deliver orders to customers using the shortest route for each vehicle, while preserving the capacity limit of each vehicle.
We have created two heuristic algorithms: one using a nearest-neighbour terminology and the other one taking inspiration of clustering close customers. We also show three custom developed Local Search algorithms. Lastly, we are going to compare our own ideas against more common used algorithms, which are tested and proved to work.

— Alexander & Narongrit

2 Custom Heuristics Algorithms

The first algorithm, which we have created, takes the approach of the Nearest Neighbour idea. Given an instance of CVRP, the starting point - the depot where the vehicles are being loaded with customer requests - is the point, where all calculations begins. We start by checking the nearest point from the depot and add it to the first route. We then add next nearest point from the point recently added, and add it to the route. We continue until we are out of capacity, which is determined by each instance. We continue adding the nearest neighbour between the points until we have reached our capacity limit. At one point, we have covered all points, covering the problem instance, and giving us one or more routes for one or more vehicles. The algorithm is described in pseudocode:

Algorithm 1 Custom CVRP Heuristic - Nearest Neighbour Approach

Require: $Point_1 \dots Point_N$
Ensure: *Solution* (solution to the CVRP instance)

```
1: function ALGORITHM(Points[ ])
2:     capacity ← 0
3:     data ← CVRP instance
4:     visited ← empty array containing visited nodes
5:     shortestdistance ← 0
6:     current ← keep track of next node to visit
7:     for i ← 0 to length of data-1 do
8:         for j ← 1 to length of data do
9:             temp ← euclideandistance(current, j)
10:            if j not in visited then
11:                if temp ≤ shortestdistance then
12:                    shortestdistance ← temp
13:                    current ← j
14:                    visited ← j
    ▷ checking if capacity requirements are met
15:                    if current total capacity ≤ max capacity then
16:                        capacity ← capacity of current node
17:                    else
18:                        Add starting point to end of route and
19:                        add capacity of current node to variable(capacity)
20:        shortestdistance ← 0                          ▷ reset shortestdistance
21:    return solution
```

The first suggestion for an algorithm has a time complexity of $O(n^2)$. The algorithm

has to iterate through the points of the instance twice. Given a point in the solution space, we are trying to find the shortest distance to any point from the current point. When we have found it, we are marking the point, and making it the new current point. Then we continue to find the shortest distance to any non-visited point, while preserving the maximum capacity of the vehicle. At some point, we are out of space in the vehicle, leaving us no choice but to return to the depot. From here we start a new route until all points are visited.
Our next algorithm is inspired by Cluster Analysis [2], which is popular in machine learning and data mining. The idea is to group a set of points lying close to each other in an euclidean space. By choosing such approach to attack the CVRP, we can ensure points are lying close to each other before we are deriving a route.
The heuristic algorithm has the following behaviour: (1) Pick a customer point furthest from away from the depot, which is not in the solution route; (2) Repeatly find nearest customer point from (1), which is not in any solution route and group such point together until the capacity is reached and save the points as a cluster; (3) Repeat (1) and (2) until we cover all customer points

The complexity of this algorithm is $O(n^2)$ since the main while loop terminates after we covered all customer points in a given instance. Furthermore each iteration in the while loop is executed in $O(n)$.

Algorithm 2 Custom CVRP Heuristic - Clustering Approach

Require: *customers*: List of customer coordinate with it's capacity
Require: *depot*: Depot Point
Require: *maxcap*: Maximum capacity for each route
Ensure: *routes*: Solution to the CVRP instance

```
routes ← empty array containing solution routes
while there is customer points not in solution do
    capacity ← 0
    route ← [depot]
    startpoint ← furthest customer point from depot which is not in solution
    add startpoint to route
    capacity ← capacity + startpoint.capacity
    while capacity < maxcap do
        closest ← closest point from startpoint not in solution
        c ← closest.capacity
        if c + capacity > maxcap then break
        capacity ← capacity + c
        add closest to route
    add route to routes
return routes
```

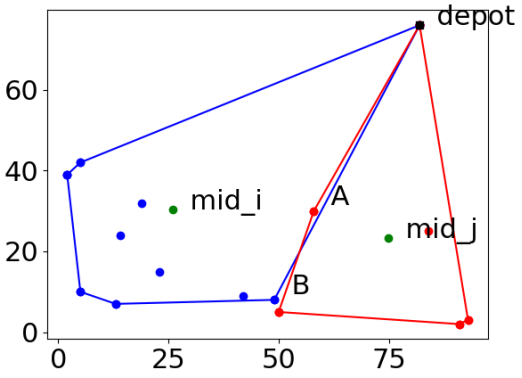
3 First Custom Local Search Algorithms

Having executed the heuristics, providing us with solutions to the initial problem instances, we can continue optimizing. This is where the Local Search Optimization algorithms comes into play. Given a canonical solution, we want to optimize it to reduce the total cost of the solution. We have provided three custom Local Search (LS) algorithms, trying to optimize the canonical solution. All of our localsearches optimize solution by make exchanges of visiting node between 2 routes in solution set. Thus they can be used in combination with original Traveling salesman problem's localsearches such as 2-opt and 3-opt, where improvements are done by swapping and reversing order of visiting nodes in single route.
The first LS algorithm looks into the generated routes done by the heuristics. It takes two routes and compare them with each other, checking if any swaps are possible in the two routes. A swap is possible if the distance is being reduced in at least one route, while still preserving the maximum capacity limit of both routes. The second LS algorithm looks like the first but with a minor change. It takes one route and compares it with every other route to find a better solution. It then continues doing this with every other route.

4 Second Custom Local Search Algorithms

This localsearch algorithm for CVRP is inspired from the fact that if we wanted to find a customer point to move from one route to another, such point has to be the point that form a convex hull of the route. We developed an algorithm to reduce the number of comparisons done by each iteration on each route by computing the convex hull points of each route, and only use those point as candidates for interchanging points in the routes. We also gain the benefit from haveing such list to estimate a "central" of the route by computing the average of convex hull point. We use the central point as the reference point of a route to find the closest point in others. These points are visualized in the figure, where lines are convex hull of a route. Point A and B are the closest points to central point from route i to route j (mid_j) and from route j to i (mid_i) respectively.

Figure 1: Convex hull points and central points of two routes

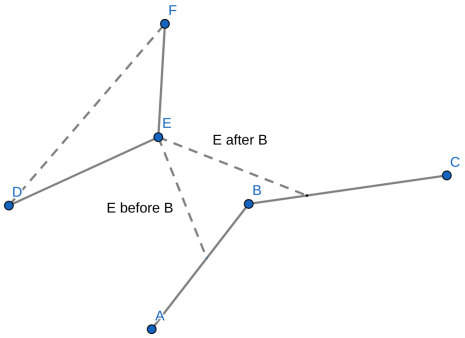


We implemented an algorithm defined by Mark de Berg [1] which has a runtime complexity of $O(nlogn)$. Our local search is given by a pair of routes: namely i and j. We find the closest point from route i to j by comparing the central points in the convex i to the central point of the convex j and closest point from j to i respectively. The possibilities of interchanging point from route i to j are defined in the following three cases:

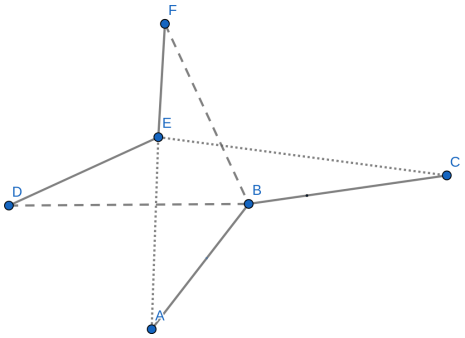
1. If the capacity constraint from moving i to j does not exceed its maximum, we compare if moving the closest customer point from i to j generates a better route. If so, we proceed and move the point from i to j
2. If the capacity constraint from moving j to i does not exceed its maximum, we compare if moving the closest customer point from j to i generates a better route. If so, we proceed and move the point from j to i

3. if the capacity constraint from swapping the closest customer point of i to j does not exceed, we compare if swapping the points of the two routes generates a better route. If so, we swap the candidate point from route i to route j and j to i respectively.

When we are performing the moving cases (1 and 2), we should also take into account the tour order. This is shown in figure 2a



(a) Move Case



(b) Swap Case

Moving point E - which is the closest point in route $j$to i - is resulted in a different tour between visiting E before B or after. Thus, it is necessary to compare cost between these two sub cases before performing the move operation as shown Appendix C, Line 9. The complexity is $O(n^2)$ and all operations inside the loop have the complexity of at most $O(n)$. If we have the improvement of route, we recompute the convex hull of 2 routes, thus, the complexity of each iteration of this localsearch is at most $O(n^3 logn)$

5 Performance Analysis & Boxplots

It has been a challenge to come up with our own custom solutions. The results we got from our algorithms and experiments were good, but we do see struggles in our design. An example of this is shown in the NN algorithm, where the first couple of routes are good, but then the later have longer distances between the points, because they are "leftovers".
Our algorithms were built without trying to be biased towards other existing algorithms. We tried to use our own imagination on how the problems could be solved. We have been inspired by concepts but not algorithms directly, and it has made us more aware on how to tackle problems and how not to.
We have tried to create boxplots for our custom algorithms to give an indication

whether or not the algorithms performs somewhat similar on the same instance, but with a bigger dataset. Meaning: it is not completely random, even though the instance looks the same but has more data. If you look at the boxplot of our custom "close-index" local search algorithm - Figure 5 - applied on the heuristic cluster algorithm and compare it to the two-opt local search algorithm - Figure 9 - applied on the same heuristic, then you are able to see the two-opt does better - especially on the golden instances - but on the other instances, we are getting there.
All the boxplots can be found in the appendices.

6 Process Analysis

The group work went well. We were both enthusiastic about the work and wanted to try out as much as possible. The communication between us has been nice and we have been splitting out the work in somewhat equal pieces to adapt to the workload from other courses.

Alexander Lerche Falk

- Develop Nearest Neighbour Approach Heuristric
- Develop Custom Local Search Algorithms 1
- Performance analysis of Heuristics and Localsearchs

Narongrit Unwerawattana

- Develop Clustering Approach Heuristic
- Develop Custom Local Search Algorithms 2
- Develop 2 and 3 opt localsearch

both

- Writing report

References

[1] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 1997.

[2] Wikipedia. Cluster analysis, March 2019.

Appendices

A
Custom Cluster Algorithm to solve CVRP

Algorithm 3 Cluster CVRP Localsearch

Require: *closest_point*$(A, points)$: returns point in points which hase minimum distance from A

Require: *elim_empty_routes*: returns list of routes where the empty routes are removed

Ensure: *routes*: Better Solution to the CVRP instance

```
1: function ALGORITHM(routes, maxcap)
2:     route_convexes is an array storing convex_hull for each route
3:     improvements ← True
4:     while improvements do
5:         for all i, j such that i, j are routes index combination do
6:             central_point_i ← average of point in convex i
7:             central_point_j ← average of point in convex j
8:             closest_from_i_j ← closest_point(mid_point_i, convex_routes[j])
9:             closest_from_j_i ← closest_point(mid_point_j, convex_routes[i])
10:            if closest_from_i_j and closest_from_j_i is Found then
11:                if closest_from_i_j fits in route j then
12:                    add move to route improvements j
13:                    make the swap
14:                else if closest_from_j_i fits in route i then
15:                    add move to route improvements i
16:                    make the swap
17:                else if capacity is not reached by swapping customers then
18:                    swap the route-points between i and j
19:                if improvements then
20:                    recalculate convex hull of route i and j
21:        if not improvements then
22:            routes = elim_empty_routes(routes) return routes
```

B LocalSearch Algorithm - Swap if any improvement

Algorithm 4 swap_if_improvement

Require: $dist(A, B)$ returns distance from point A to B

1: **function** SWAP_IF_IMPROVEMENT($route\_i, route\_j, j\_to\_i, i\_to\_j$)
2: `A, B, C` $\leftarrow$ `route_i[i_to_j-1], route_i[i_to_j], route_i[i_to_j+1 mod(%) len(route_i)]`
3: `D, E, F` $\leftarrow$ `route_j[j_to_i-1], route_j[j_to_i], route_j[j_to_i+1 mod(%) len(route_j)]`
4: `old_distance_i` $\leftarrow dist($`A, B`$) + dist($`B, C`$)$
5: `old_distance_j` $\leftarrow dist($`D, E`$) + dist($`E, F`$)$
6: `new_distance_i` $\leftarrow dist($`A, E`$) + dist($`E, C`$)$
7: `new_distance_j` $\leftarrow dist($`D, B`$) + dist($`B, F`$)$
8: **if** `old_distance_i + old_distance_j > new_distance_i + new_distance_j` **then**
swap point B of route i with point E of route j
return True

C LocalSearch Algorithm - Move if any improvement

Algorithm 5 move_if_improvement

Require: $dist(A, B)$ returns distance from point A to B

function MOVE_IF_IMPROVEMENT($route\_i, route\_j, i\_to\_j, j\_to\_i$)
    A, B, C ← route_i[i_to_j -1], route_i[i_to_j], route_i[i_to_j+1 % len(route_i)]
    D, E, F ← route_j[j_to_i -1], route_j[j_to_i], route_j[j_to_i+1 % len(route_j)]
    old_distance_i ← $dist$(A, B) + $dist$(B, C)
    old_distance_j ← $dist$(D, E) + $dist$(E, F)
    new_dist_before_B ← $dist$(A, E) + $dist$(E, B) + $dist$(B, C)
    new_dist_after_B ← $dist$(A, B) + $dist$(B, E) + $dist$(E, C)
    new_distance_j ← $dist$(D, F)
    if new_dist_before_B < $new\_dist\_after\_B$ **then**
        if old_distance_i + old_distance_j > new_dist_before_B + new_distance_j **then**
            move point E from route j to route i before B
    else
        if old_distance_i + old_distance_j > new_dist_after_B + new_distance_j **then**
            move point E from route j to route i after B

D BOXPLOT FIGURES

Figure 2: Heuristic NN Algorithm - Custom

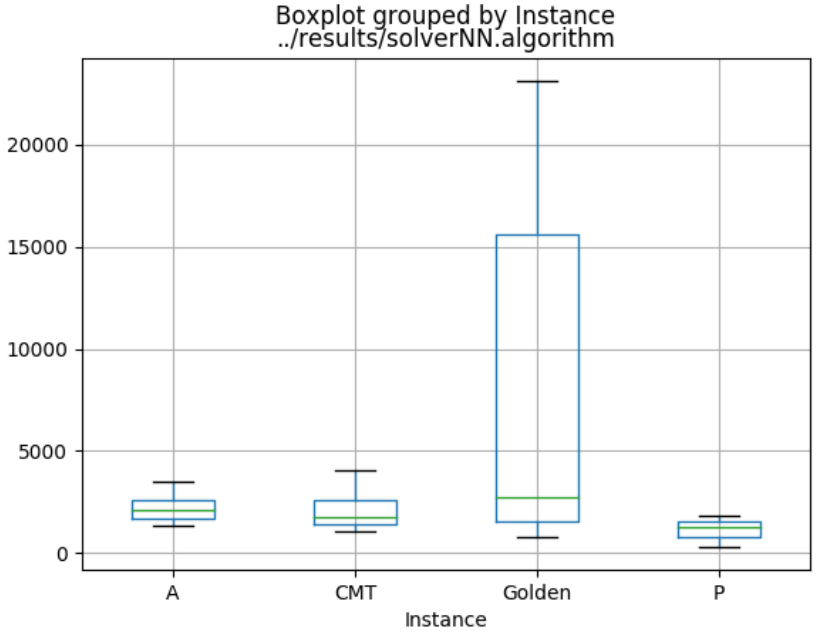


Figure 3: Heuristic Cluster Algorithm - Custom

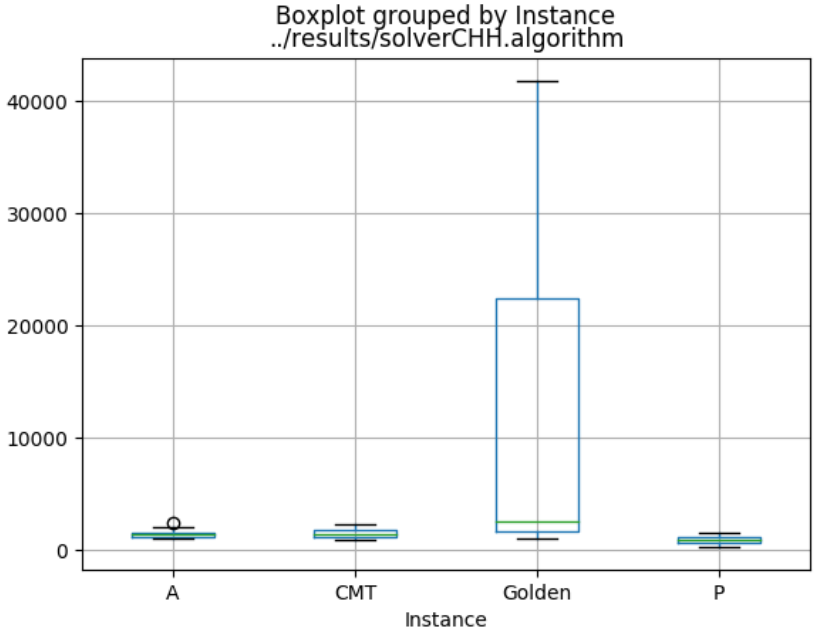


Figure 4: Local Search Algorithm - Custom - Close index route for NN heustic

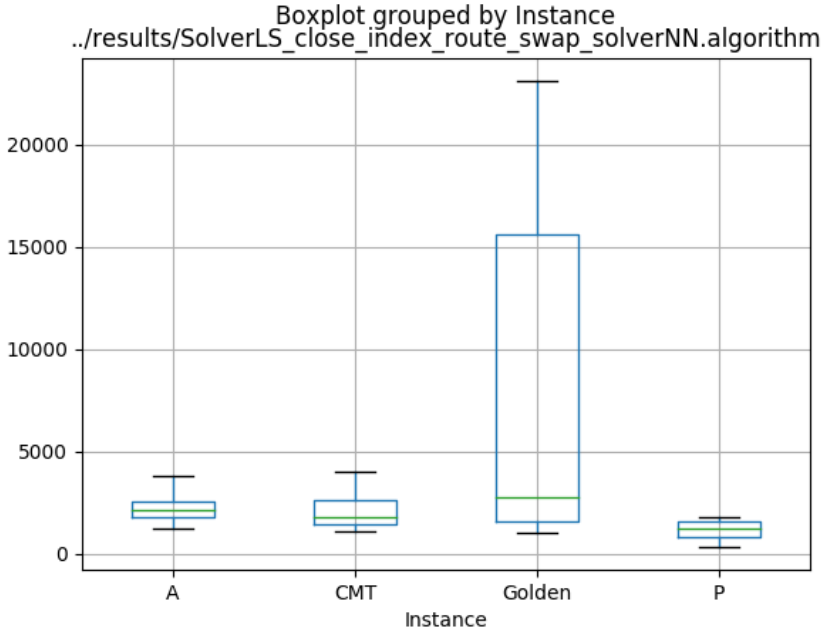


Figure 5: Close Index Custom Local Search Figure for cluster heuristic

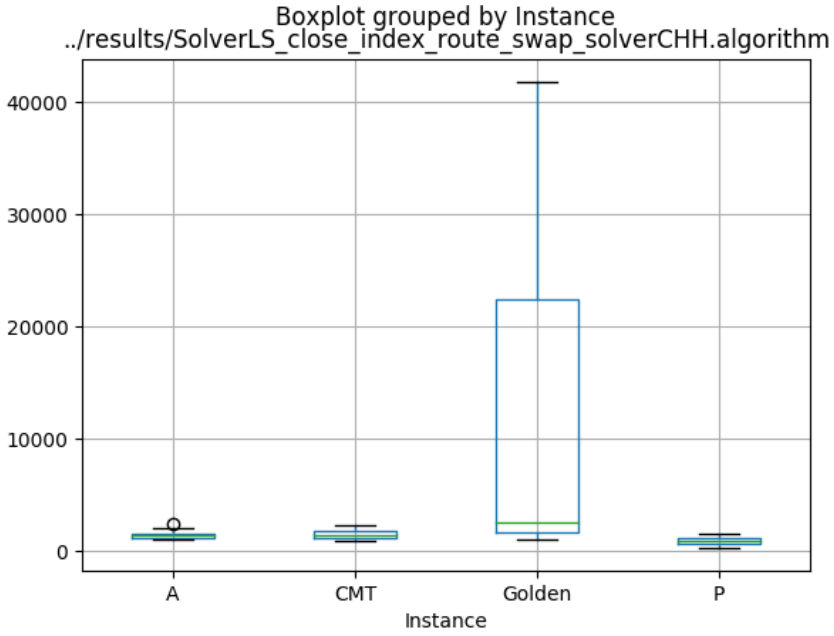


Figure 6: Local Search Algorithm - Custom - All index combinations route for NN heuristic

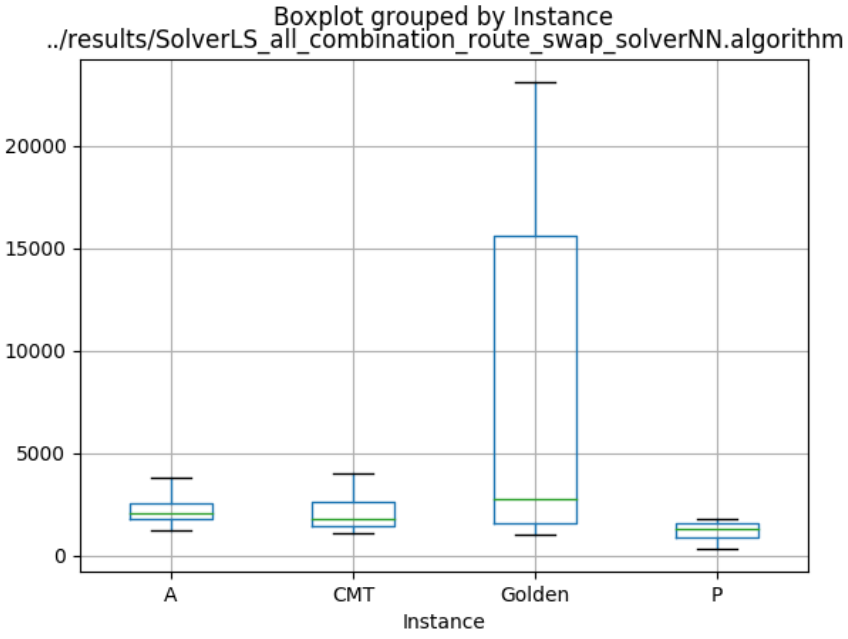


Figure 7: Local Search Algorithm - Custom - All index combinations route for cluster heuristic

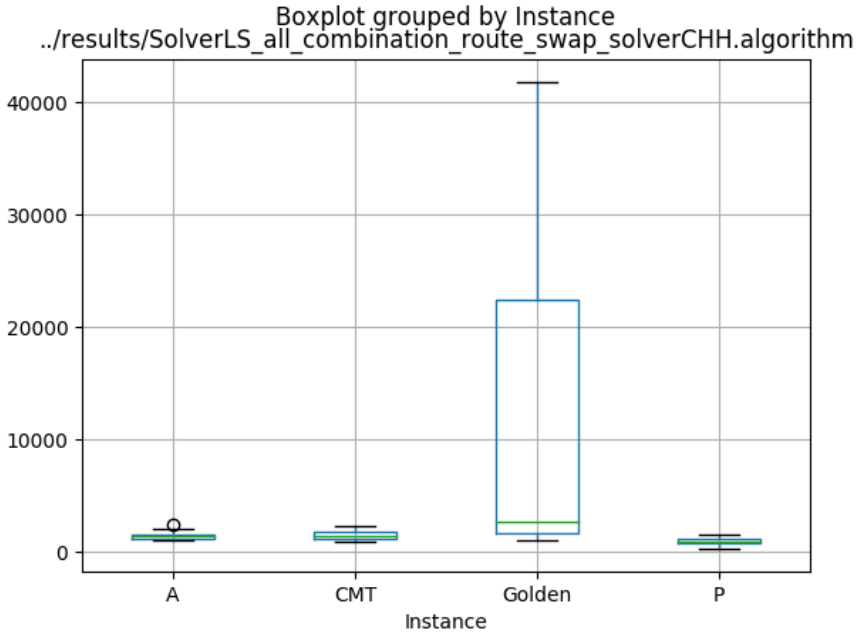


Figure 8: Local Search Algorithm - Known - Two Opt for NN heuristic

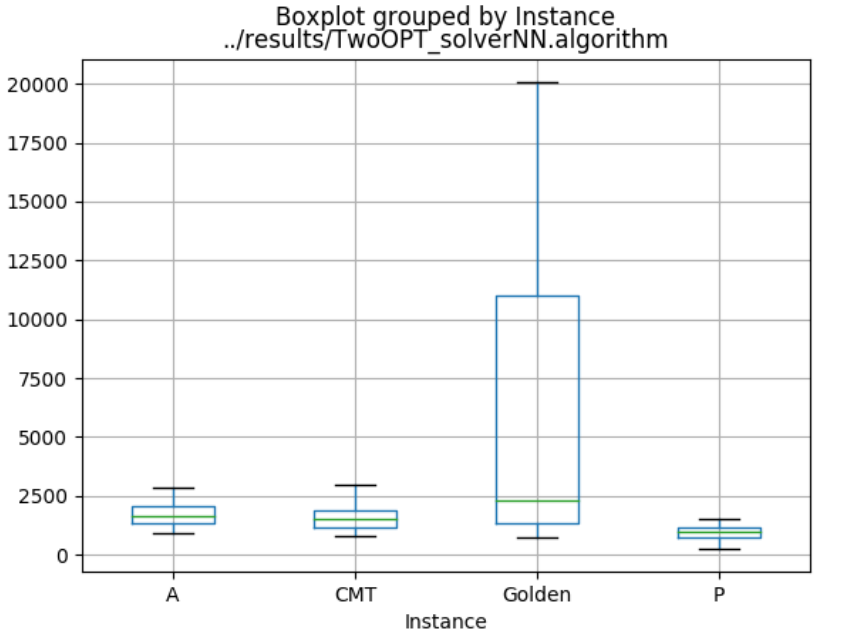


Figure 9: Two Opt Known Local Search Figure for cluster heuristic